UNIT III

GREEDY AND DYNAMIC PROGRAMMING

Introduction - Greedy: Huffman Coding - Knapsack Problem - Minimum Spanning Tree (Kruskals Algorithm). Dynamic Programming: 0/1 Knapsack Problem - Travelling Salesman Problem - Multistage Graph- Forward path and backward path.

INTRODUCTION

Greedy Method

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum.
2. **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems.

The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Applications

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later.

Examples of such greedy algorithms are

- Kruskal's algorithm for finding minimum spanning trees
- Prim's algorithm for finding minimum spanning trees
- Huffman coding Algorithm for finding optimum Huffman trees.
- Used in Networking too. Greedy algorithms appear in network routing as well. Using greedy routing, a message is forwarded to the neighboring node which is "closest" to the destination. The notion of a node's location (and hence "closeness") may be determined by its physical location, as in geographic routing used by ad hoc networks. Location may also be an entirely artificial construct as in small world routing and distributed hash table.

HUFFMAN ENCODING

Huffman codes are very effective and widely used technique for compressing data. Huffman encoding problem is of finding the minimum length bit string which can be used to encode a string of symbols. It uses a table of frequencies of occurrence of each character to represent each character as a binary string, optimally. It uses a simple heap based priority queue. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree.

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-legth codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream. Huffman's greedy algorithm looks at the occurrence of each character and it as a binary string in an optimal way.

There are mainly two major parts in Huffman Coding

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Constructing a Huffman tree

A greedy algorithm that constructs an optimal prefix code called a Huffman code. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |c| leaves and perform |c|-1 "merging" operations to create the final tree.

Pseudo Code

Data Structure used: Priority queue = Q

Huffman (c)

```
n = |c|
Q = c
for i =1 to n-1
  do z = Allocate-Node ()
       x = left[z] = EXTRACT_MIN(Q)
       y = right[z] = EXTRACT_MIN(Q)
       f[z] = f[x] + f[y]
       INSERT (Q, z)
return EXTRACT_MIN(Q)
```

Analysis

- Q implemented as a binary heap.
- line 2 can be performed by using BUILD-HEAP in O(n) time.
- FOR loop executed |n| - 1 times and since each heap operation requires O(lg n) time.
  - ⇨ the FOR loop contributes (|n| - 1) O(lg n)
  - ⇨ O(n lg n)
- Thus the total running time of Huffman on the set of n characters is O(nlg n).

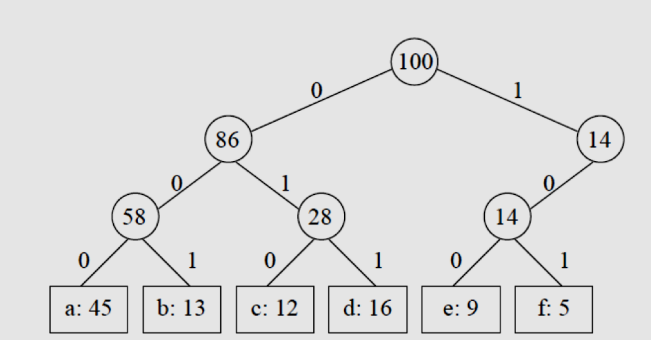
Example

Suppose we have a data consists of 100,000 characters that we want to compress. The characters in the data occur with following frequencies.

	A	B	C	D	E	F
Frequency	45,000	13,000	12,000	16,000	9,000	5,000

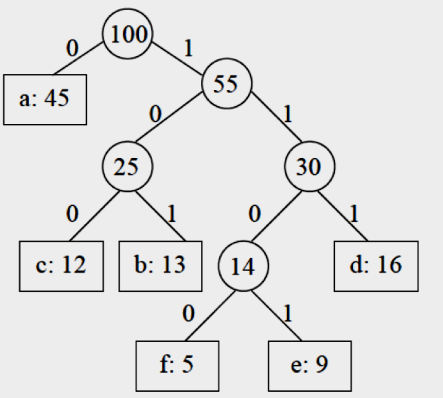
Fixed Length Code : In fixed length code, needs 3 bits to represent six(6) characters.

	A	B	C	D	E	F
Frequency	45,000	13,000	12,000	16,000	9,000	5,000
Fixed Length code	000	001	010	011	100	101



- Total number of characters are 45,000 + 13,000 + 12,000 + 16,000 + 9,000 + 5,000 = 1000,000.
- Add each character is assigned 3-bit codeword => 3 * 1000,000 = 3000,000 bits.

Variable code Length



	A	B	C	D	E	F
frequency (in thousands)	45000	13000	12000	16000	9000	5000
fixed-length Code	0	1	10	11	100	101
variable-length code	0	101	100	111	1101	1100

- requires 45×1+13×3+12×3+16×3+9×4+5×4=224,000 bits

Practice Problems

1. Find the Huffman code for the following message "COLLEGE OF ENGINEERING"
2. Consider the following set of frequencies A=2, B=5, C=7, D=8, E=7, F=22, G=4, H=17. Find Huffman code for the same.
3. Consider the following set of frequencies A=8, B=15, C=37, D=18, E=47, F=22, G=20, H=17. Find Huffman code for the same.

KNAPSACK PROBLEMS - Fractional knapsack

There are n items in a store. For i =1,2, . . . , n, item i has weight $w_i > 0$ and worth $v_i > 0$. Thief can carry a maximum weight of W pounds in a knapsack. In this version of a problem the items can be broken into smaller piece, so the thief may decide to carry only a fraction x_i of object i, where $0 \le x_i \le 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load.

Applications

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatory, computer science, complexity theory, cryptography and applied mathematics.

Algorithm

Greedy-fractional-knapsack (*w, v, W*)

```
FOR i =1 to n
   do x[i] =0
weight = 0
while weight < W
   do i = best remaining item
      IF weight + w[i] ≤ W
         then x[i] = 1
            weight = weight + w[i]
         else
            x[i] = (w - weight) / w[i]
            weight = W
return x
```

Analysis

If the items are already sorted into decreasing order of v_i / w_i, then the while-loop takes a time in *O(n)*;

Therefore, the total time including the sort is in *O(n log n)*.

Note:

- The greedy algorithm that always selects the **most valuable object** does **not always** find an optimal solution to the Fractional Knapsack problem.
- The greedy algorithm that always selects the **lighter object** does **not always** find an optimal solution to the Fractional Knapsack problem.
- Theorem: The greedy algorithm that always selects the **object with better ratio value/weight always finds an optimal solution** to the Fractional Knapsack problems.
- A greedy algorithm for an optimization problem **always makes the choice that looks best at the moment** and adds it to the current sub solution. What's output at the end is an optimal solution.
- Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available.

MINIMUM SPANNING TREE (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight that all other spanning trees of the same graph. In real world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

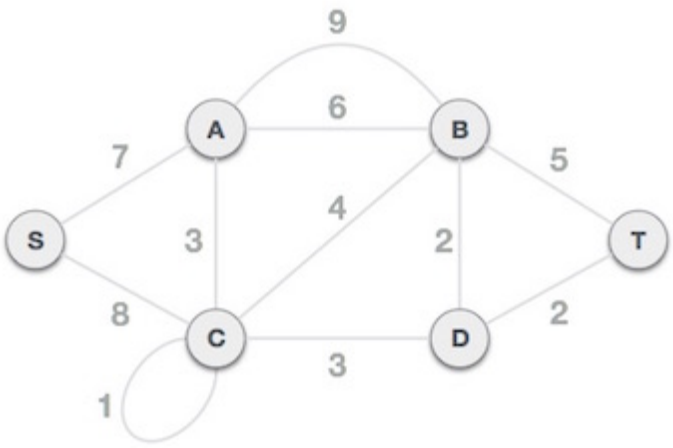
Kruskal's Algorithm

Kruskal's algorithm to find minimum cost spanning tree uses greedy approach. This algorithm treats the graph as a forest and every node it as an individual tree. A tree connects to another only and only if it has least cost among all available options and does not violate MST properties.

Algorithm

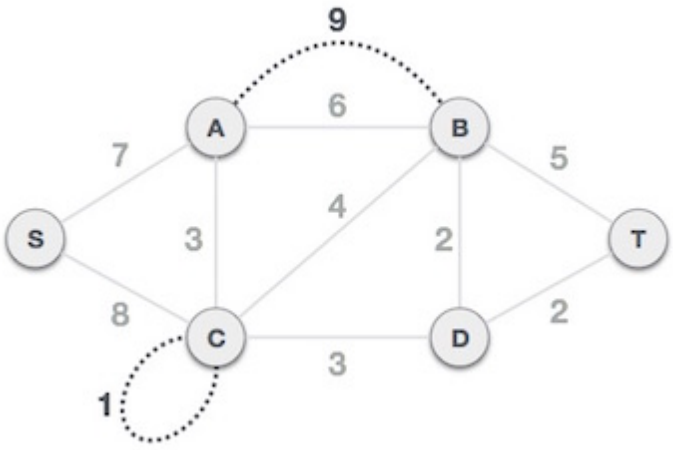
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

To understand Kruskal's algorithm we shall take the following example –

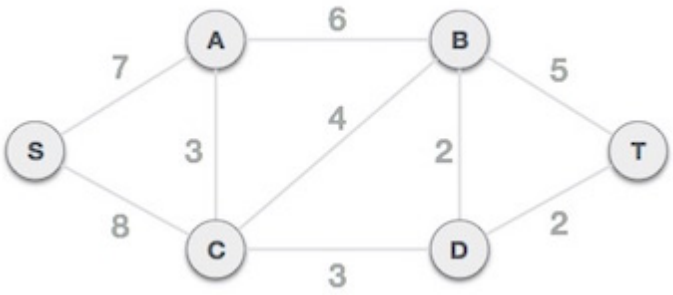


Step 1 - Remove all loops & Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has least cost associated and remove all others.

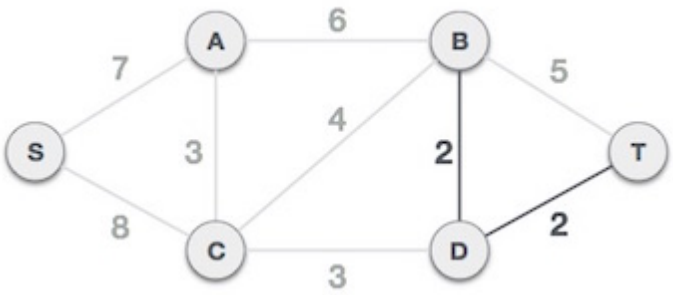


Step 2 - Arrange all edges in their increasing order of weight: Next step is to create a set of edges & weight and arrange them in ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

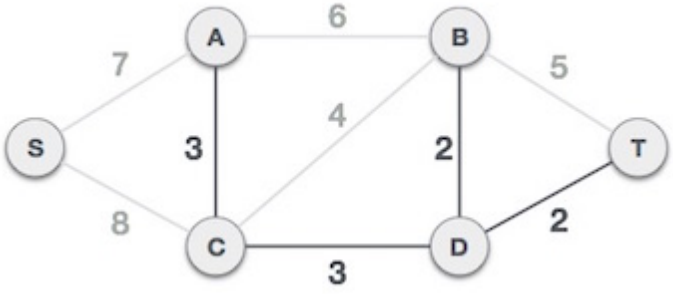
Step 3 - Add the edge which has least weightage

Now we start adding edges to graph beginning from the one which has least weight. At all time, we shall keep checking that the spanning properties are remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in graph.

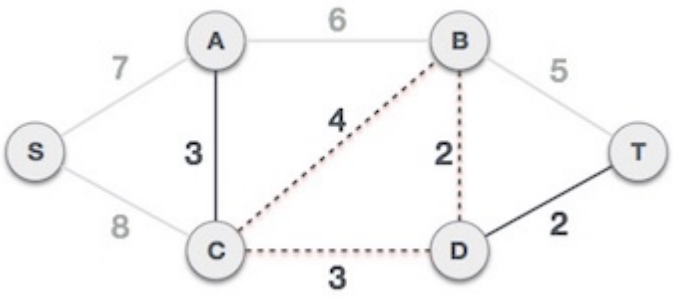


The least cost is 2 and edges involved are B,D and D,T so we add them. Adding them does not violate spanning tree properties so we continue to our next edge selection.

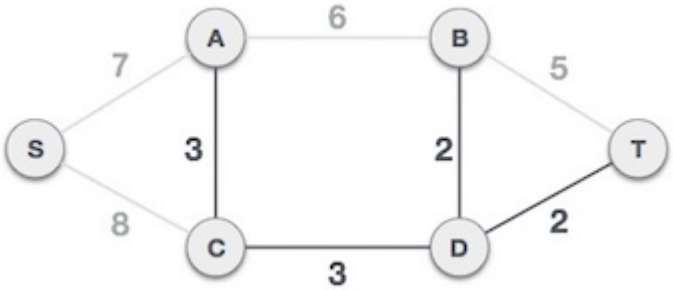
Next cost is 3, and associated edges are A,C and C,D. So we add them –



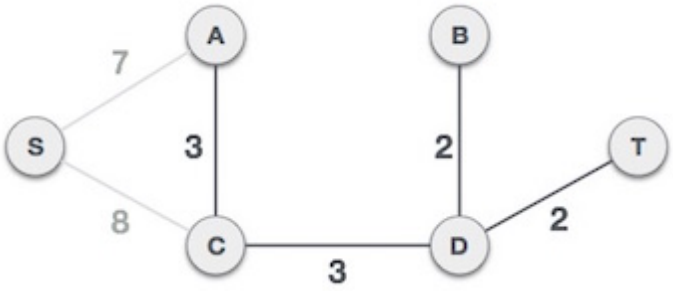
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph



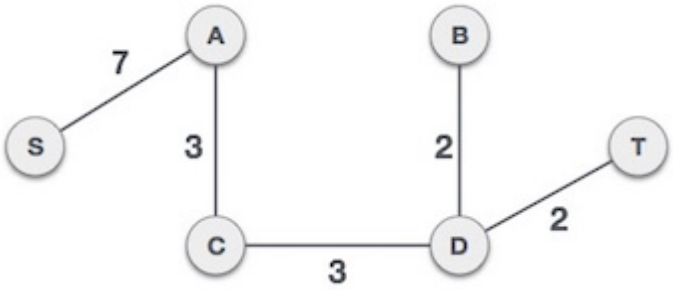
...and we ignore it. And in the process we shall ignore/avoid all edges which create circuit.



We observe that edges with cost 5 and 6 also create circuits and we ignore them and move on.



Now we are left with only one node to be added. Between two least cost edges available 7, 8 we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we have MST

Time complexity

O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calls minHeapify(). So, overall complexity is O(nlogn).

DYNAMIC PROGRAMMING:

It is used when the solution can be recursively described in terms of solutions to subproblems (optimal substructure). Algorithm finds solutions to subproblems and stores them in memory for later use. More efficient than "brute-force methods", which solve the same subproblems over and over again.

Optimal substructure:

Optimal solution to problem consists of optimal solutions to subproblems

Overlapping subproblems:

Few subproblems in total, many recurring instances of each

Bottom up approach:

Solve bottom-up, building a table of solved subproblems that are used to solve larger ones

Greedy vs Dynamic Programming

Greedy method	Dynamic Programming
make an optimal choice (without knowing solutions to subproblems) and then solve remaining subproblems	solve subproblems first, then use those solutions to make an optimal choice
solutions are top down	solutions are bottom up
Best choice does not depend on solutions to subproblems.	Choice at each step depends on solutions to subproblems
Make best choice at current time, then work on subproblems. Best choice does depend on choices so far	Many subproblems are repeated in solving larger problems. This repetition results in great savings when the computation is bottom up
Optimal Substructure: solution to problem contains within it optimal solutions to subproblems	**Optimal Substructure**: solution to problem contains within it optimal solutions to subproblems
Fractional knapsack: at each step, choose item with highest ratio	0-1 Knapsack: to determine whether to include item i for a given size, must consider best solution, at that size, with and without item i

0/1 KNAPSACK PROBLEM

The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W,

maximize $\sum_{i=1}^{n} v_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0,1\}$.

Here x_i represents the number of instances of items i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

Optimal substructure:

To consider all subsets of items, there can be two cases for every item:

(1) the item is included in the optimal subset,

(2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

(1) Maximum value obtained by n-1 items and W weight (excluding nth item).

(2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

Pseudo Code

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
for j from 0 to W do:
        m[0, j] := 0
for i from 1 to n do:
        for j from 0 to W do:
                if w[i] <= j then:
                    m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
                else:    m[i, j] := m[i-1, j]
```

Analysis

This solution will therefore run in $O(nW)$ time

Example

Input: 5 objects, C = 100

W	10	20	30	40	50
V	20	30	66	40	60

Solve the knapsack problem **using both greedy and dynamic approach**

Solution

Fractional Knapsack

Input: 5 objects, C = 100

W	10	20	30	40	50
V	20	30	66	40	60

Solution

Given

Total no of items = 5, sack capacity = 100 ,

1) **Step 1** : Find the Value/weight ratio

Ratio = $\frac{v_i}{w_i}$

Object	1	2	3	4	5
Ratio = $\frac{v_i}{w_i}$	2	1.5	2.2	1	1.2

2) **Step 2 :** Sort the items according to the ratio and Select the item according to its highest ratio

i) First item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1				

Sack Weight = 30 <100

Sack value = 2.2 * 30 = 66

ii) Second item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1			

Sack Weight = 30 +10 = 40 <100

Sack value = 66 + (2*10) =86

iii) Third item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1	1		

Sack Weight = 40 +20 = 60 <100

Sack value = 86 + (1.5*20) = 116

iv) Fourth item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1	1	1	

Sack Weight = 60 +50 = **110 >100 Hence item 4 is selected partially.**

Sack Weight = 60+(100-60) = **100 <=100**

Sack value = 116 + (1.2*40) = 116+48= 164

Now the sack is FULL. Hence we stop

Total selected weight 100 and total value = $2.2 * 30 + 2 * 10 + 1.5 * 20 + 1.2 * 40 = 164$.

Total value = 164

0-1 Knapsack

Input: 5 objects, C = 100

W	10	20	30	40	50
V	20	30	66	40	60

Solution:

Given

Total no of items = 5, sack capacity = 100 ,

1) **Step 1** : Find the Value/weight ratio

Ratio = $\frac{v_i}{w_i}$

Object	1	2	3	4	5
Ratio = $\frac{v_i}{w_i}$	2	1.5	2.2	1	1.2

2) **Step 2 :** Sort the items according to the ratio and Select the item according to its highest ratio

i) First item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1				

Sack Weight = 30 <100

Sack value = 2.2 * 30 = 66

ii) Second item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1			

Sack Weight = 30 +10 = 40 <100

Sack value = 66 + (2*10) =86

iii) Third item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1	1		

Sack Weight = 40 +20 = 60 <100

Sack value = 86 + (1.5*20) = 116

iv) Fourth item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1	1	1	

Sack Weight = 60 +50 = **110 >100 Hence item 4 is skipped and item 5 is selected**

v) Fifth item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = $\frac{v_i}{w_i}$	2.2	2	1.5	1.2	1
Selected item	1	1	1	0	1

Sack Weight = 60+40 = **100 <=100**

Sack value = 116 + (1*40) = 156

Now the sack is FULL. Hence we stop. Total value = 156

Difference between Fractional and o/1 knapsack problem

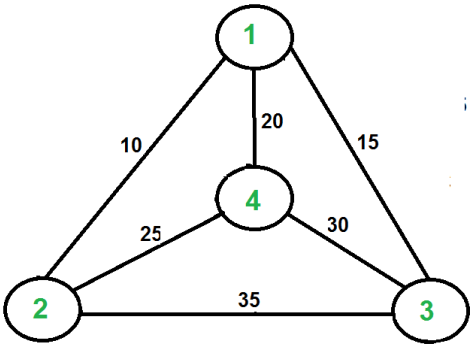
0-1 Knapsack	Fractional Knapsack
N items (can be the same or different)	N items (can be the same or different)
Must **leave or take** (ie 0-1) each item (eg ingots of gold)	an take **fractional part** of each item (eg bags of gold dust)
Dynamic programming works, greedy does not	Greedy works and DP algorithms work

TRAVELLING SALESMAN PROBLEM

Travelling Salesman Problem (TSP): *Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.*

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every **city exactly once**. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a **minimum weight Hamiltonian Cycle**.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.

Time Complexity: ?(n!)

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,....n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?

To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.

We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},
 C(S, i) = dist(1, i)
Else if size of S is greater than 2.
 C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

Example

Distance matrix:

$$C = \begin{pmatrix} 0 & 2 & 9 & 10 \\ 1 & 0 & 6 & 4 \\ 15 & 7 & 0 & 8 \\ 6 & 3 & 12 & 0 \end{pmatrix}$$

g(2,Ø) = c21 = 1

g(3,ø) = c31 = 15

g(4,ø) = c41 = 6

k = 1, consider sets of 1 element:

Set {2}: g(3,{2}) = c32 + g(2,ø) = c32 + c21 = 7 + 1 = 8 p(3,{2}) = 2

g(4,{2}) = c42 + g(2,ø) = c42 + c21 = 3 + 1 = 4 p(4,{2}) = 2

Set {3}: g(2,{3}) = c23 + g(3,ø) = c23 + c31 = 6 + 15 = 21 p(2,{3}) = 3

g(4,{3}) = c43 + g(3,ø) = c43 + c31 = 12 + 15 = 27 p(4,{3}) = 3

Set {4}: g(2,{4}) = c24 + g(4,ø) = c24 + c41 = 4 + 6 = 10 p(2,{4}) = 4

g(3,{4}) = c34 + g(4,ø) = c34 + c41 = 8 + 6 = 14 p(3,{4}) = 4

k = 2, consider sets of 2 elements:

Set {2,3}: g(4,{2,3}) = min {c42 + g(2,{3}), c43 + g(3,{2})} = min {3+21, 12+8}= min {24, 20}= 20

p(4,{2,3}) = 3

Set {2,4}: g(3,{2,4}) = min {c32 + g(2,{4}), c34 + g(4,{2})} = min {7+10, 8+4}= min {17, 12} = 12

p(3,{2,4}) = 4

Set {3,4}: g(2,{3,4}) = min {c23 + g(3,{4}), c24 + g(4,{3})} = min {6+14, 4+27}= min {20, 31}= 20

p(2,{3,4}) = 3

Length of an optimal tour:

f = g(1,{2,3,4}) = min { c12 + g(2,{3,4}), c13 + g(3,{2,4}), c14 + g(4,{2,3}) }

= min {2 + 20, 9 + 12, 10 + 20} = min {22, 21, 30} = 21

Successor of node 1: p(1,{2,3,4}) = 3

Successor of node 3: p(3, {2,4}) = 4

Successor of node 4: p(4, {2}) = 2

Optimal TSP tour: 1–> 3 –> 4 –> 2 –>1

MULTISTAGE GRAPH

"**Dynamic programming** is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions"

Minimum spanning of multistage graph using dynamic programming

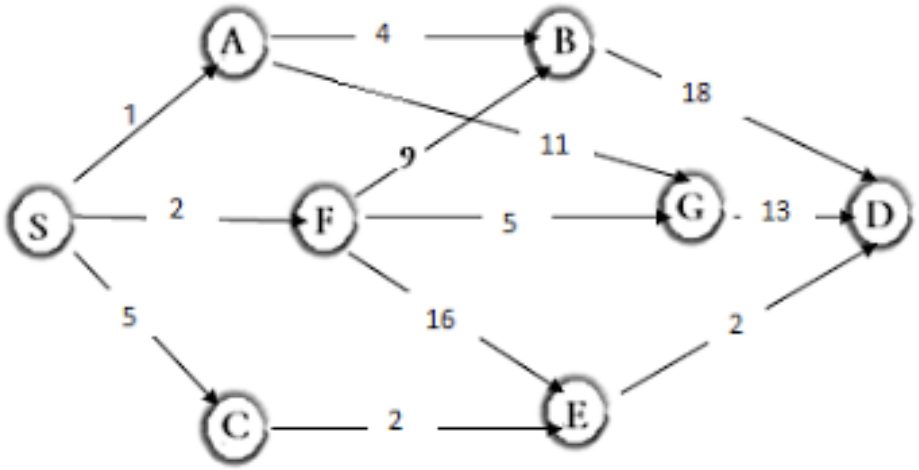
a. Forward Approach
b. Backward Approach

a. <u>Forward Approach</u>:

Spanning a multiple stage graph using following considerations

- Identify source and destination nodes.
- Find all possible paths to reach destination from source and sum of weights of adjacent nodes.
- The path giving the least weight will be the minimum spanning path.

Consider a multistage graph given below



Identifying source and destination nodes.

Source node -> S

Destination node -> D

The possible ways to connect S & D

d(S,D) = min { 1 + d(A,D) ; 2 + d(F,D) ; 5 + d(C,D) } (1)

d(A,D) = min{ 4 + d(B,D) ; 9 + d(G,D) }

= min{ 4 + 18 ; 9 + 13 } 'Substation weights

= min{ 22 ; 22 }

d(A,D) = 22 (2)

d(F,D) = min{ 9 + d(B,D) ; 5 + d(G,D) ; 16 + d(E,D) }

= min{ 9 + d(B,D) ; 5 + d(G,D) ; 16 + d(E,D) }

= min{ 9 + 18 ; 5 + 13 ; 16 + 2} 'Substation weights

= min{ 27 ; 18 ; 18}

d(F,D) = 18 (3)

d(C,D) = min{ 2 + d(E,D) }

= min{ 2 + 2} 'Substation weights

= min{ 4}

d(C,D) = 4 (4)

substitution of 2,3,4 in 1 gives

d(S,D) = min { 1 + d(A,D) ; 2 + d(F,D) ; 5 + d(C,D) }

d(S,D) = min { 1 + 22 ; 2 + 18 ; 5 + 4 }

d(S,D) = min { 23 ; 20 ; 9 }

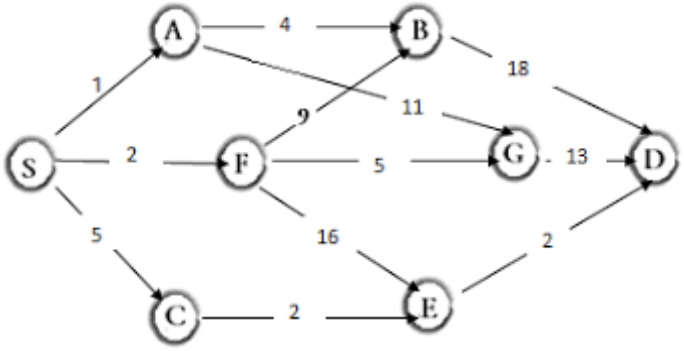
d(S,D) = 9

Hence according to Forward Approach minimum spanning path from S to D is S -> C -> E -> D

b. <u>Backward Approach</u>:

Backward Approach is just the reverse of forward approach, here Source node and the next node is considered at every stage.

Considering same Multi staged Graph,



1 -> 2

Source node S to next nodes A, F and C

d(S,A) = 1

d(S,F) = 2

d(S,C) = 5

Source node S to next nodes B, G and E

d(S,B) = min{ 1 + d(A,B) ; 2 + d(F,B)}

= min{ 1 + 4 ; 2 + 9}

= min{ 5 ; 11}

d(S,B) = 5

d(S,G) = min{ 1 + d(A,G) ; 2 + d(F,G)}

= min{ 1 + 11 ; 2 + 5}

= min{ 12 ; 7}

d(S,G) = 7

d(S,E) = min{ 2 + d(F,E) ; 5 + d(C,E)}

= min{ 1 + 16 ; 5 + 2}

= min{ 17 ; 7}

d(S,E) = 7

d(S,D) = min{ 5 + d(B,D) ; 7 + d(G,D) ; 7 + d(E,D)}

= min{ 5 + 18 ; 7 + 13 ; 7 + 2}

= min{ 23 ; 20 ; 9}

d(S,D) = 9

Hence according to Backward Approach minimum spanning path from S to D is **S -> C -> E -> D**

1. Dynamic Programming algorithms need to store the results of intermediate sub problems.
    a. True
    b. False
2. The knapsack problem belongs to the domain of _____ problems.
    a. Optimization
    b. NP Complete
    c. Linear Solution
    d. Sorting
3. Suppose we have three items as below and knapsack capacity is 50. Item value (Pi,Wi) = (60,10), (100, 20),(120,30).The optimal solution is to pick
    a. Items 1 and 2
    b. Items 1 and 3
    c. Items 2 and 3
    d. Items 1,2 and 3
4. _____ technique is for solving problems with overlapping sub problems.
    a. Divide and Conquer
    b. Greedy Techniques
    c. Back tracking
    d. Dynamic Programming
5. Which is applying Dynamic Programming to non optimization problems
    a. Memory functions
    b. Binomial coefficients
    c. Washall
    d. Greedy method
6. Which of the following algorithm computes the transitive closure of a directed graph
    a. Warshall algorithm
    b. Floyd algorithm
    c. Dijkstra algorithm
    d. Huffman coding
7. Which of the following standard algorithms is not a Greedy algorithm?
    a. Dijkstra's shortest path algorithm
    b. Bellmen Ford Shortest path algorithm
    c. Kruskal algorithm
    d. Huffman Coding

8. A networking company uses a compression technique to encode the message before transmitting over the network. Suppose the message contains the following characters with their frequency:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

If the compression technique used is Huffman Coding, how many bits will be saved in the message?

   a. 224
   b. 800
   c. 576
   d. 324

9. What is the time complexity of Huffman Coding?
   a. O(N)
   b. O(NlogN)
   c. O(N(logN)^2)
   d. O(N^2)
10. Which of the following is true about Huffman Coding.
   a. Huffman coding may become lossy in some cases
   b. Huffman Codes may not be optimal lossless codes in some cases
   c. In Huffman coding, no code is prefix of any other code.
   d. All of the above
11. Suppose the letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. Which of the following is the Huffman code for the letter a, b, c, d, e, f?
   a. 0, 10, 110, 1110, 11110, 11111
   b. 11, 10, 011, 010, 001, 000
   c. 11, 10, 01, 001, 0001, 0000
   d. 110, 100, 010, 000, 001, 111
12. Suppose the letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. What is the average length of Huffman codes?
   a. 3

b. 2.1875
c. 2.25
d. 1.19375

13. Memory function uses
    a. Bottom up approach and top down
    b. Branch and bound
    c. Dynamic approach
    d. Greedy approach
14. _____is a connected graph which connected acyclic subgraphs that contains all the vertices of the graph
    a. Spanning tree
    b. Weighted graph
    c. D-Graph
    d. Binary search tree
15. _____ constructs the MST through the sequence of expanding subtrees
    a. Kruskal
    b. Prims
    c. Both a and b
    d. Dijkstra
16. Which of the following is not a greedy method
    a. Prims
    b. Kruskal
    c. Dijkstra
    d. Warshal
17. Which coding tree is updated each time when a new character read from the source text
    a. Huffman coding
    b. Dynamic huff man coding
    c. Variable length coding
    d. Fixed length coding
18. What is the complexity for the knapsack by dynamic approach
    a. O(nW)
    b. O(n+W)
    c. O(W)
    d. O(n)

19. Distance matrix can be generated using the following algorithm
    a. Dijkstra
    b. Prims
    c. Kruskal
    d. Floyd
20. Principles of Optimality is used in
    a. Greedy approach
    b. Dynamic Approach
    c. Divide and conquer
    d. Back tracking

4 marks Questions

1. Write short notes on dynamic programming.
2. State the general principles of greedy technique.
3. Compare greedy algorithm and Dynamic programming.
4. What is the need for dynamic programming? Explain with example.
5. Define Huffman codes? List out the applications for Huffman codes.
6. Explain the Knapsack problem with an example.
7. Justify Greedy approach for solving knack problem.

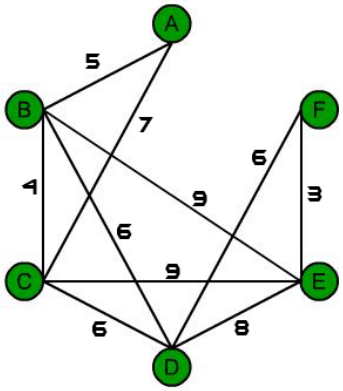
12 marks questions

1. Find the Huffman code for the following message “COLLEGE OF ENGINEERING”
2. Consider the following set of frequencies A=2, B=5, C=7, D=8, E=7, F=22, G=4, H=17. Find Huffman code for the same.
3. Consider the knapsack instance n=3, m=20, (w1,w2,w3) = (18,15,10), (p1,p2,p3) = (25,24,15). Solve using greedy and dynamic programming concept.

In dynamic programming, the output to stage n become the input to *

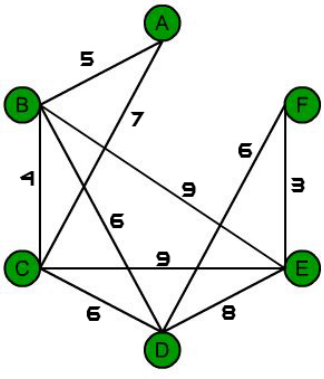
- stage n-1
- stage n itself
- stage n+1
- stage n-2

Refer to the graph of distance (in km) between cities at http://goo.gl/kXe2J. The shortest route from City A to City F has length *



- 15 km
- 22 km
- 17 km
- None of the above

Refer to the graph of distance (in km) between cities at http://goo.gl/kXe2J. If you HAVE to take the route from City D to City E, the shortest route from City A to City F has length *

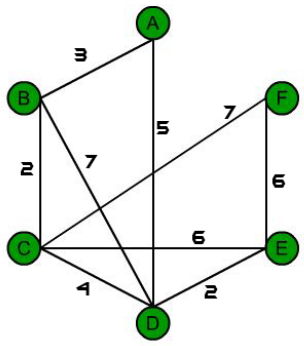


- 17 km
- 22 km
- 20 km
- None of the above

For the above question, what will be the shortest route? *

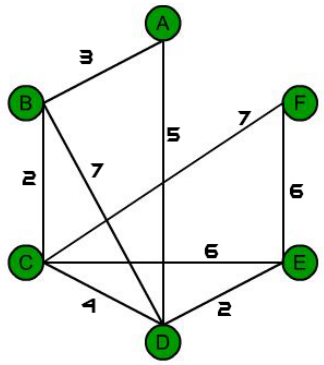
- A -> B -> D -> E -> F
- A -> C -> D -> E -> F
- A -> C -> E -> F
- A -> B -> D -> F

Refer to the network at at http://goo.gl/KJPV2. The shortest route from Node A to City F has length *



- 13
- 12
- 7
- None of the above

Refer to the network at at http://goo.gl/KJPV2. The shortest route from Node A to City F is *



- A -> B -> D -> E -> F
- A -> D -> E -> F
- A -> B -> C -> F
- None of the above

Which of the following algorithms CANNOT be said to employ elements of dynamic programming? *

- Bellman-Ford Algorithm
- Dijkstra's Shortest Path
- Floyd Warshall Algorithm
- Bubble Sort Algorithm

Which statement is true? *

- If a dynamic-programming problem satisfies the optimal-substructure property, then a locally optimal solution is globally optimal
- If a greedy choice property satisfies the optimal-substructure property, then a locally optimal solution is globally optimal
- Both of above
- None of above